Online Multimedia

Winter Semester 2019/20

Tutorial 01 – Web Programming Introduction

Today's Agenda

- Syllabus
- Web Programming Introduction
- Roundup Quiz

Syllabus

Tutorials Team

Florian Bemmann
florian.bemmann@ifi.lmu.de

Changkun Ou
changkun.ou@ifi.lmu.de

Thomas Weber
thomas.weber@ifi.lmu.de

Aleksa Ri
aleksa.ri@protonmail.com

AS

André Schmidt

RL

Ruben Lenz

Tutorial Times

- Tutorials are for **Major students**

Day	Time	Tutor
Monday	16 – 18 h	André
Monday	18 – 20 h	André
Wednesday	16 – 18 h	Aleksa

- There are dedicated tutorials for Minor students in the "Multimedia im Netz" lecture

Tutorials – Why are we doing this?

- Application and **immersion** of lecture content
- **Hands-on** activities and discussion
- Opportunity to discuss and ask **questions**
- **Preparation** of the upcoming assignment
- **Discussion** of the solutions to exercises
- Preparation for work / job.

Procedure – Part 1

- Slides and assignment online prior to tutorial
- Due dates for assignments: one week.
  Wednesday to **Wednesday**
- News, updates, and important announcements on the official website:
  http://www.medien.ifi.lmu.de/lehre/ws1920/omm/

Procedure – Part 2

- Doing the assignments is completely **voluntary**.
- We recommend you do the assignments.
  - They're fun and challenging.
  - They go beyond the lecture content.
  - They prepare you to pass the exam.
  - Statistics show that you get better grades if you do the assignments
- Assignments are turned in via UniWorX
  - Make sure to check the due date
  - You can't hand in an assignment after the deadline.
  - Individual- or group submission

Sample Solutions & Material

- We do **not** provide sample solutions.
- If you submit your solution to UniWorX, you will get proper feedback and can improve your solution on your own.

- Materials & example solutions:
  - Code from the tutorials is always on GitHub: https://github.com/mimuc/omm-ws1920
  - You are welcome to **fork** that repository and share **your own** solutions for the assignments in the /assignments/solutions/ subfolders.
    There's a How-to in the README.md

Office Hours: For the details

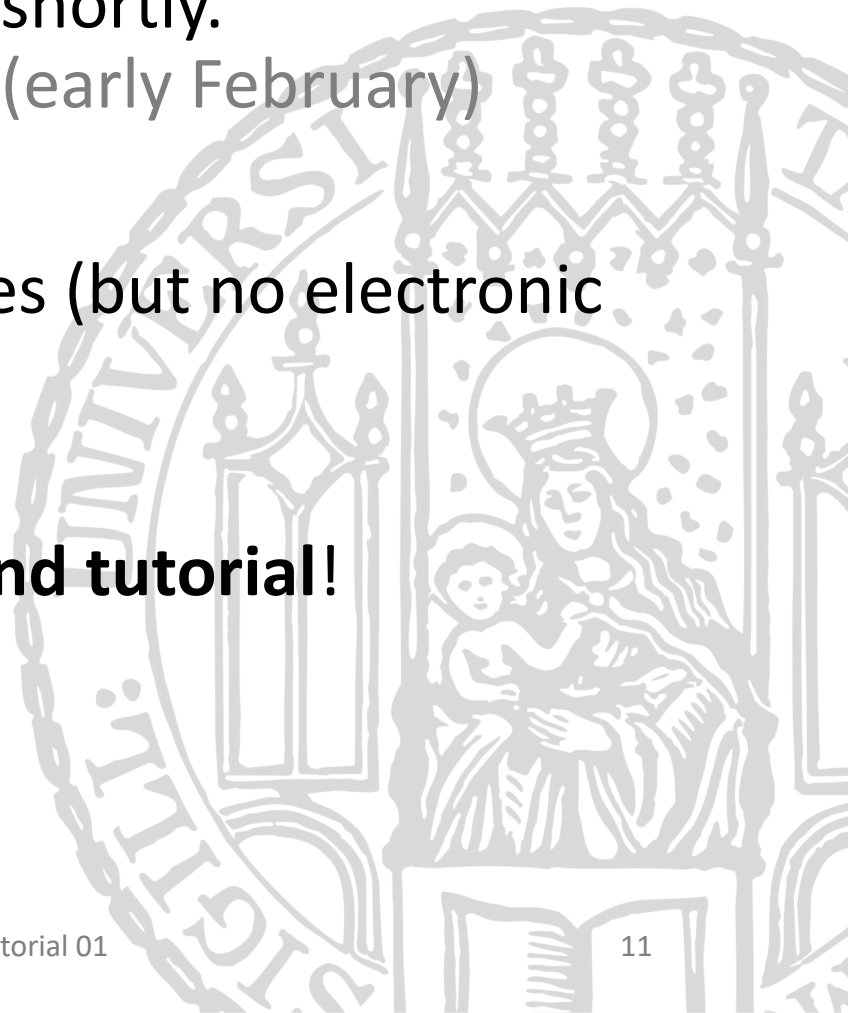
- Friday between 16:00 and 17:00
- Modality
  - 1:1 with teaching assistant
  - around 25-30 minutes
- How-to
  - send an email to one of the tutorial supervisors at least one day in advance
  - send a list of questions via email in advance
  - no requests for solutions
- This is an opportunity to talk about details and things you didn't catch either in the tutorial

Exam

- Final date, time, and location will be announced shortly.
  Probably the first week of “vorlesungsfreie Zeit” (early February)

- Open Book: You can bring print-outs, books, notes (but no electronic devices)

- The exam includes tasks from both the **lecture and tutorial**!
  Approximately 40-50% tutorial and assignments
  (especially code and concepts)



Tutorial Plan (subject to change)

Date	Topics	Technology Focus
21.10.	Web Programming Introduction	Orga, JavaScript
28.10.	AJAX	JavaScript, AJAX
04.11.	Advanced JavaScript	JavaScript, TypeScript, Event Loop, Bundling
11.11.	WebComponents	JavaScript, React
18.11.	React	JavaScript, React
25.11.	Frontend Testing & Performance	Linting, Testing, CI
02.12.	Backend Programming	
09.12.	REST APIs	
16.12.	Web-Architecture	
23.12. 30.12. 08.01.		Christmas Break
13.01.	NoSQL Databases	
20.01.	Database Integration	
27.01.	Backend Testing & Performance	
03.02.	Repetition	

Tools, tools, tools

https://github.com/mimuc/omm-1920#required-toolkit

Online Multimedia - Winter Semester 2019/2020

In this repository you can find materials for the Online Multimedia lecture at LMU Munich in the Winter Semester 2019/2020.

The lecture is targeted at Informatics and Media Informatics Master students. For more details, see the course website.

Required To

To do the Break-O install these tools *as soon as possible*.

- Text Editor / Web IDE - *choose one* -
  - VS Code
  - Atom
  - Sublime
  - WebStorm (Students are eligible for a free version)

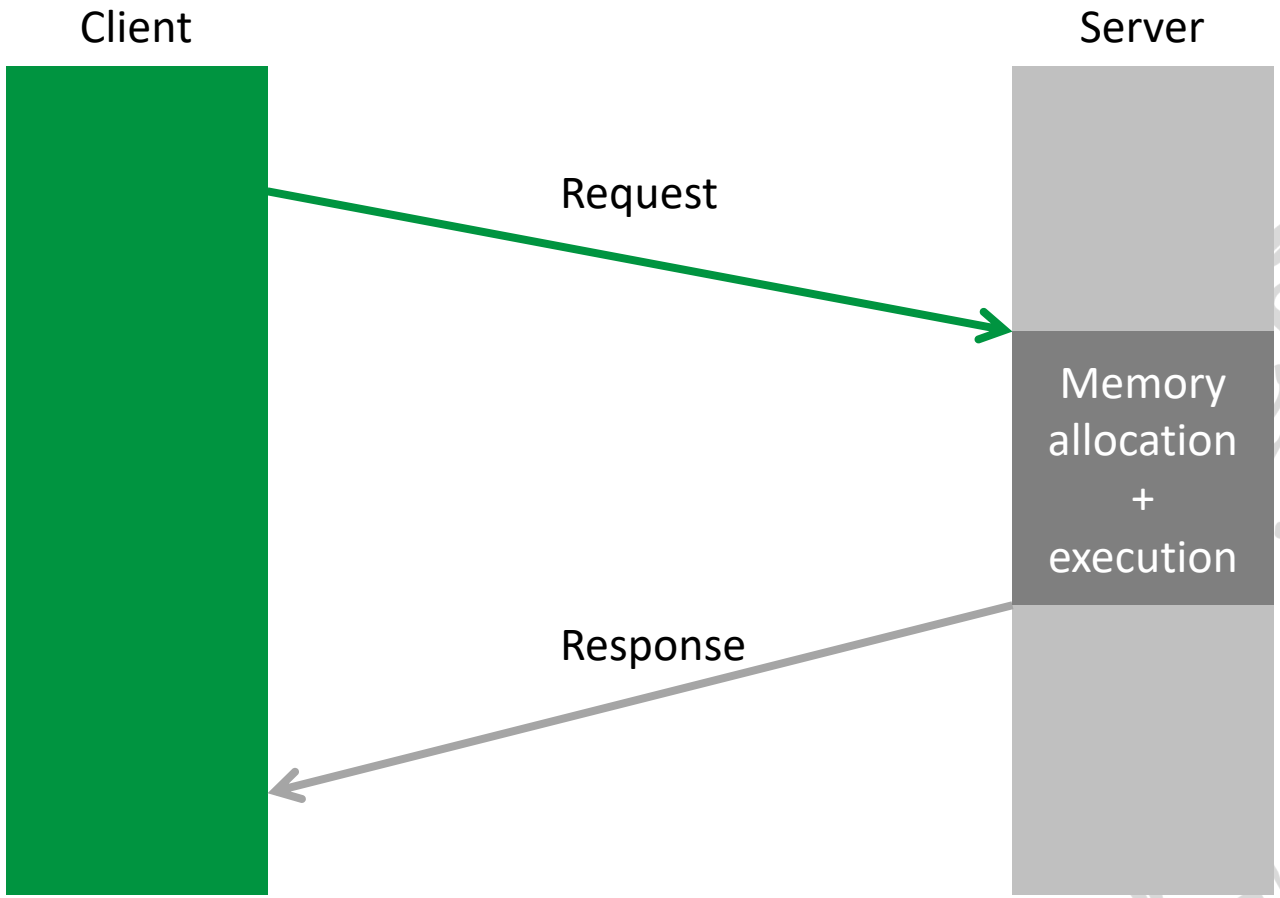
Major Subject

- Git. On Windows you need to install git from https://git-scm.com/. It's already included on macOS and Linux. On Mac you might want to install the XCode Command Line Tools to make sure you get the latest version.
  - After you're all set with git, go straight ahead to this tutorial, if you don't know git.
  - Watch this video to get you all up and running with git.
  - We recommend generating an SSH key and cloning this repository via SSH.

Be prepared!

Web-Programming Introduction

Client-Server



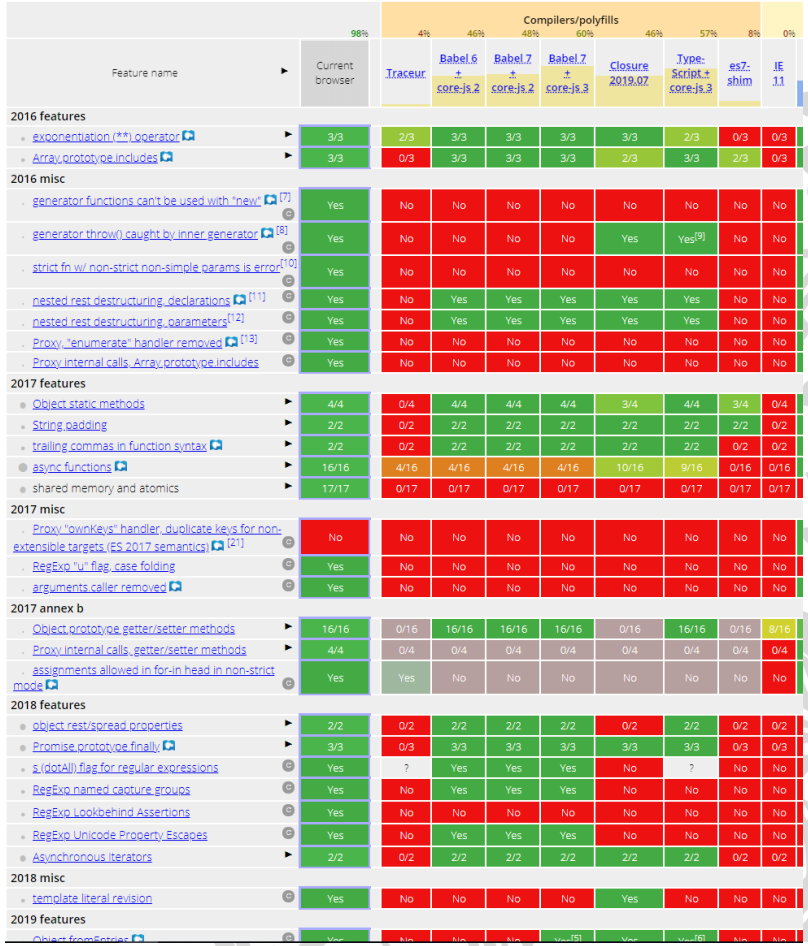
Web-Programming

- ***Frontend*** (Client)
  - HTML → Structure
  - CSS → Style
  - JavaScript → Behavior and content

- ***Backend*** (Server)
  - PHP, NodeJS, Ruby, Python…
  - Storage, authentication, hardcore calculations…

JavaScript vs. EcmaScript

- EcmaScript are all languages that comply with the ECMA-262 specification
- JavaScript is the most commonly known implementation
- Most major browsers support EcmaScript Version 6 aka ES2015
- There are, however, differences between browsers for newer or non-ES features
- One way to check whether you can use a feature: https://www.caniuse.com



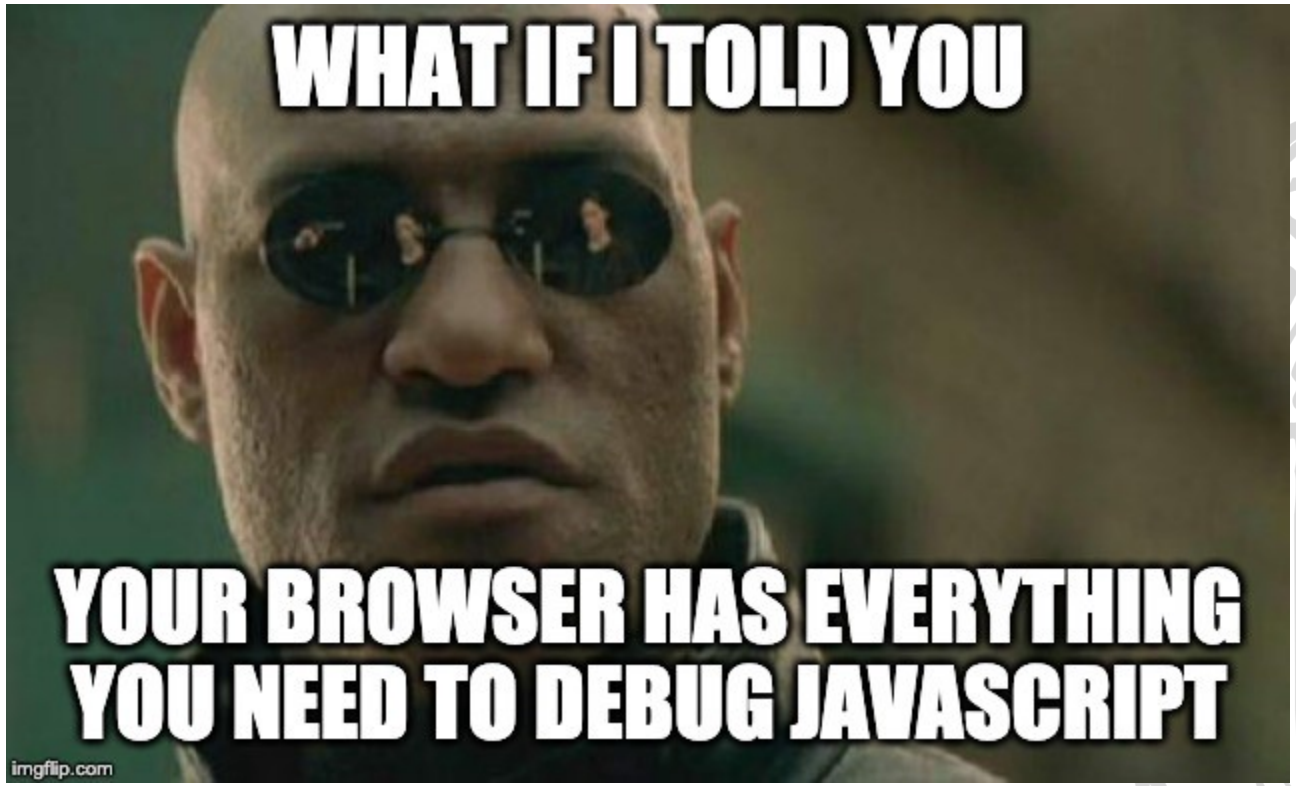
Feature name	Current browser (98%)	Traceur (4%)	Babel 6 + core-js 2 (46%)	Babel 7 + core-js 2 (48%)	Babel 7 + core-js 3 (60%)	Closure 2019.07 (46%)	TypeScript + core-js 3 (57%)	es7-shim (8%)	IE 11 (0%)
2016 features									
exponentiation (**) operator	3/3	2/3	3/3	3/3	3/3	3/3	2/3	0/3	0/3
Array.prototype.includes	3/3	0/3	3/3	3/3	3/3	2/3	3/3	2/3	0/3
2016 misc									
generator functions can't be used with "new" [7]	Yes	No	No	No	No	No	No	No	No
generator.throw() caught by inner generator [8]	Yes	No	No	No	No	Yes	Yes[9]	No	No
strict fn w/ non-strict non-simple params is error [10]	Yes	No	No	No	No	No	No	No	No
nested rest destructuring, declarations [11]	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No
nested rest destructuring, parameters [12]	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No
Proxy, "enumerate" handler removed [13]	Yes	No	No	No	No	No	No	No	No
Proxy internal calls, Array.prototype.includes	Yes	No	No	No	No	No	No	No	No
2017 features									
Object.static methods	4/4	0/4	4/4	4/4	4/4	3/4	4/4	3/4	0/4
String padding	2/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2
trailing commas in function syntax	2/2	0/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2
async functions	16/16	4/16	4/16	4/16	4/16	10/16	9/16	0/16	0/16
shared memory and atomics	17/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17
2017 misc									
Proxy "ownKeys" handler, duplicate keys for non-extensible targets (ES 2017 semantics) [21]	No	No	No	No	No	No	No	No	No
RegExp "u" flag, case folding	Yes	No	No	No	No	No	No	No	No
arguments.caller removed	Yes	No	No	No	No	No	No	No	No
2017 annex b									
Object.prototype getter/setter methods	16/16	0/16	16/16	16/16	16/16	0/16	16/16	0/16	8/16
Proxy internal calls, getter/setter methods	4/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4
assignments allowed in for-in head in non-strict mode	Yes	Yes	No	No	No	No	No	No	No
2018 features									
object rest/spread properties	2/2	0/2	2/2	2/2	2/2	0/2	2/2	0/2	0/2
Promise.prototype.finally	3/3	0/3	3/3	3/3	3/3	3/3	3/3	0/3	0/3
s (dotAll) flag for regular expressions	Yes	?	Yes	Yes	Yes	No	?	No	No
RegExp named capture groups	Yes	No	Yes	Yes	Yes	No	No	No	No
RegExp Lookbehind Assertions	Yes	No	No	No	No	No	No	No	No
RegExp Unicode Property Escapes	Yes	No	Yes	Yes	Yes	No	No	No	No
Asynchronous Iterators	2/2	0/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2
2018 misc									
template literal revision	Yes	No	No	No	No	Yes	No	No	No
2019 features									
Object.fromEntries	Yes	No	No	No	Yes[5]	Yes	Yes[6]	No	No

https://kangax.github.io/compat-table/es2016plus/

JavaScript Basics

- JavaScript is a general purpose scripting language most commonly found on the web
- Some features:
  - Dynamic typing
  - Prototype-based OOP
  - First-class functions
  - Massive eco-system
- JavaScript's syntax is very similar to many other curly-brace languages

Debugging JavaScript

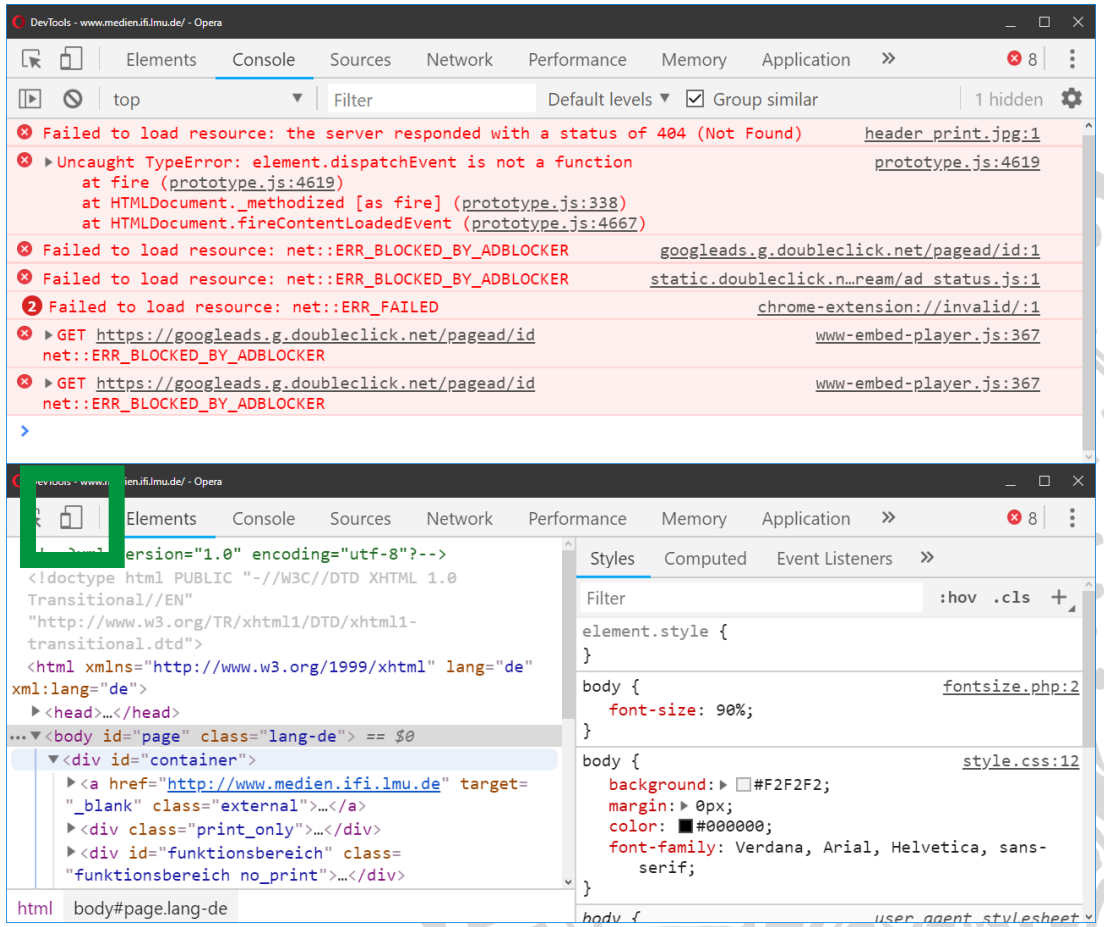


Debugging JavaScript

- **Use your browsers developer tools!**

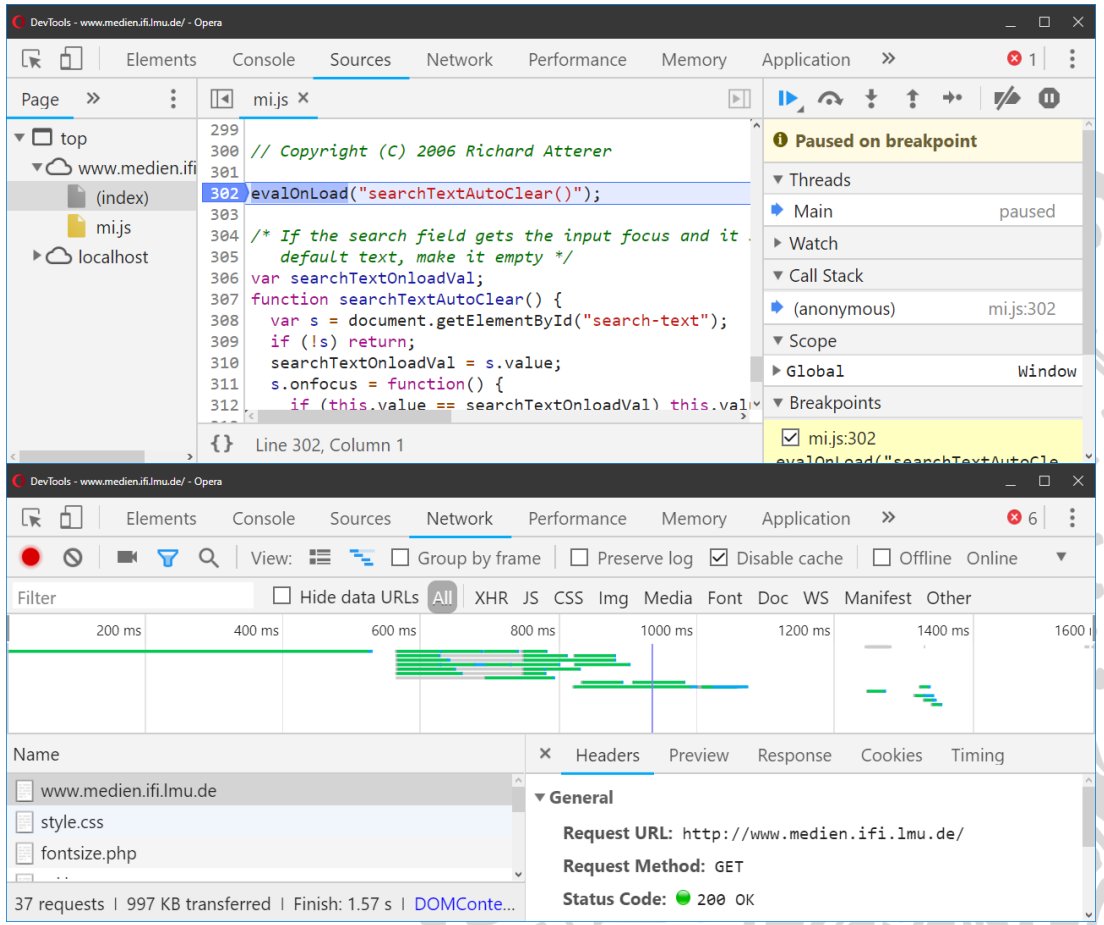
- **Console**: JavaScript console
  - All errors/outputs etc. are shown here
  - Allows manualy execution of JS snippets

- **Elements**: Page Inspector
  - Select and manipulate individual DOM nodes
  - Perfect for testing styling and whether your DOM manipulation JS did what it should
  - Can simulate other devices viewports for simple responsive testing



Debugging JavaScript

- **Sources**: JavaScript inspector
  - Allows debugging with breakpoints
- **Network**: Network Monitor
  - See what the page sends/receives
  - Network/AJAX debugging
  - Allows simulated throttling for testing with slow connections (e.g. 3G)
- Performance/Memory: Profiling
- etc.



Debugging JavaScript

What to do, when something does not work?

- Always check the JavaScript console for errors!
- **Ideally:** Set breakpoints and step through the code like you would in any other language
- *Realistically:* Use the console for print-debugging (use only for small problems)

```
function complicatedFunction () {
  var x = 0;
  // something complicated happens

  console.log('X is', x);

  // more complicated things

  console.error('Print stracktrace');
}
```

JavaScript Functions

- Functions can be defined either using the `function` keyword or as fat-arrow functions

```
function foo  (bar) { ... }
const foo = (bar) => { ... }
```

- Fat arrows are a shorter syntax for functions with some minor differences
  - Fat arrow functions have no own this, super, prototype, etc.
  - Therefore fat arrow function cannot be used as methods or constructors
  - Instead they are ideal for callbacks or a functional programming style

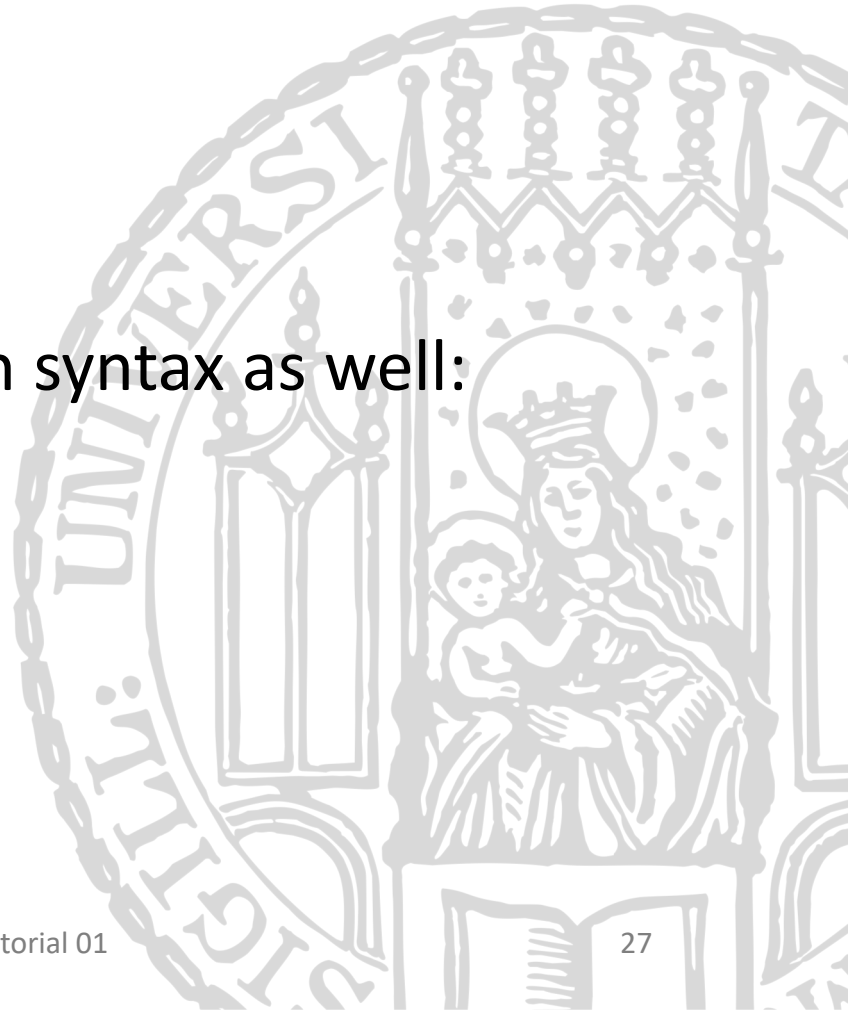
- *Btw.:* Function parameters can have default values:

```
(a, b = [ ], c = 5) => { … }
```

Objects & Arrays

Objects & Arrays

- Objects in JS can created via constructors
  `new Image()`
  or by object literal syntax:
  `{ key: value, key2: value2 }`
- Arrays are just special objects but have their own syntax as well:
  `[ 1, 2, 3, 4 ]`



Object Destructuring

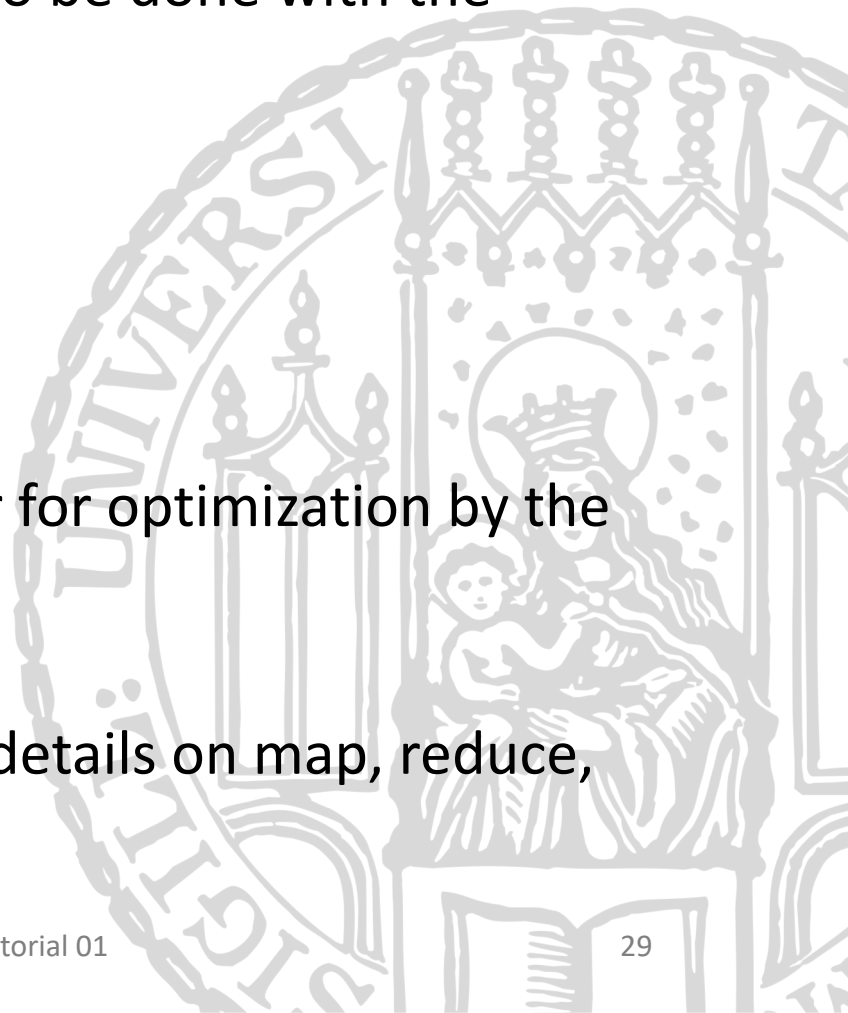
- JavaScript allows destructuring objects and arrays in a pattern-matching-like fashion:
  ```
  var { foo, bar, baz } = obj;
  ```
  is equivalent to assigning `var foo = obj.foo` & bar/baz analogously

- The spread operator ... allows using an array or object where individual values are expected:
  ```
  var arr = [ 2, 10 ];
  Math.pow(...arr) // = Math.pow(2, 10);
  ```
  ```
  const obj = { a: 1, b: 2 };
  const newObj = { ...obj, c: 3 }
  ```

- For more examples, see the appendix

Iteration

- Many array operations traditionally done in loops can also be done with the functional/callback-base alternative:
  - map: To make changes to every element
  - forEach: To perform an action for every element
  - filter: To select a subset of all elements
  - reduce: To accummulate or aggregate elments
- This often makes code simpler, easier to read, and better for optimization by the browser/runtime.
- See lecture on functional programming (e.g. ProMo) for details on map, reduce, filter etc.



Map & Reduce

```
var numbers = [ 10, 20, 30, 40, 50 ];

function duplicateValue (num) {
    return num * 2;
}

function sumElements (sumSoFar, currentArrayElement) {
  return sumSoFar + currentArrayElement;
}

const sumOfMultiples = numbers //      [ 10,  20,  30,  40,   50 ]
  .map(duplicateValue)         //      [ 20,  40,  60,  80,  100 ]
  .reduce(sumElements, 0);     // 300 = 20 + 40 + 60 + 80 + 100
```

JavaScript Scope

Scoping

- Scope in JavaScript can at times be counter-intuitive
- There are three levels of scoping: global, function and block
- Variables declared without a declaration keyword (e.g. `var`) are global scope
- Variables declared with the `var` keyword are bound in the next higher function or, if not in a function, globally

```
var iAmGlobal = 42;

function foo () {
  var iAmScopedInFoo = 12;
  iAmGlobalToo = true;
  if (true) {
    var iAmScopedInFooToo;
  }
}
```

Hoisting

- The JavaScript `var` keyword does not use block-scoping
- Variables declared with var can be used before they are declared

- This is due to hoisting:
  - variable decaration are hoisted, i.e. moved to the top of the current scope
  - "current scope" is either the parent function (function scope) or the parent script (global scope).
  - if/loop blocks are not considered scoping-blocks for hoisting

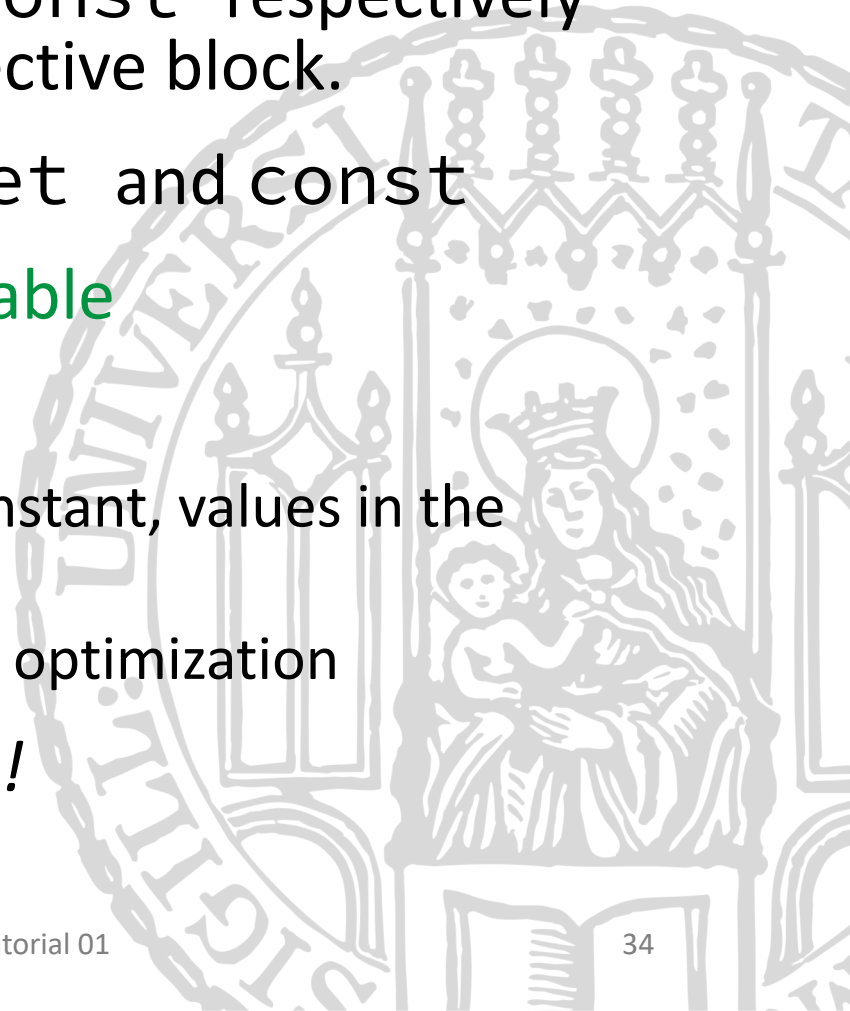
```
console.log(x);
var x = 5;

// actually is

var x;
console.log(x);
x = 5;
```

`let` and `const`

- Variables and constants declared with `let` or `const` respectively are not hoisted but instead scoped in their respective block.
- if/loop blocks are considered scope-blocks for `let` and `const`
- This usually makes their behaviour more predictable
- Values declared `const` are actual constants
  - Note that for objects, only the object-reference is constant, values in the object are still mutable
  - This allows better static checking and more agressive optimization
- *We recommend using `const` whenever possible!*



Breakout #1

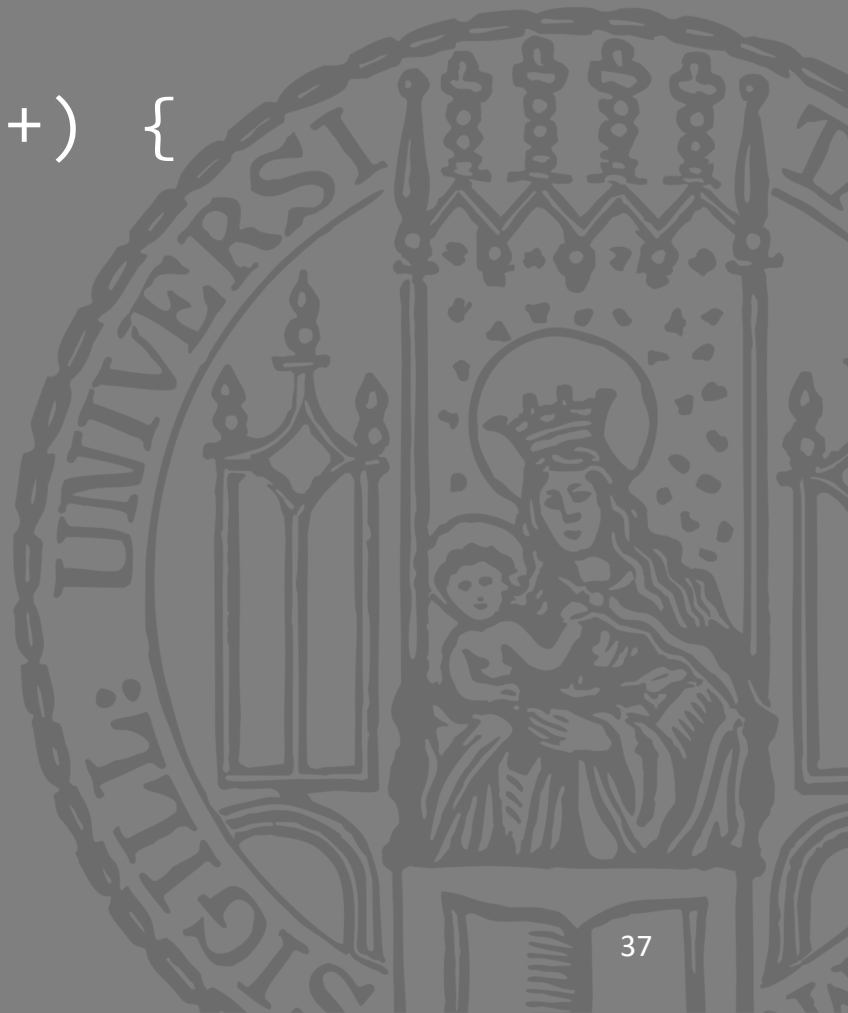
- Shorten the following code snippets as much as you can

Breakout #1

```
var arr = new Array(3);
arr[0] = 2;
arr[1] = 4;
arr[2] = 8;
arr[3] = 16;
```

Breakout #1

```
var arr = [ 2, 4, 8, 16 ];
for (var i = 0; i < arr.length; i++) {
  arr[i] = coolFunction(arr[i]);
}
```



Breakout #1

```
function myFunction ( obj ) {
  var foo = obj.foo;
  var bar = obj.bar;
  if (foo) {
    return bar;
  } else {
    return null;
  }
}
```

Promises & Async

Beware the Callback Hell

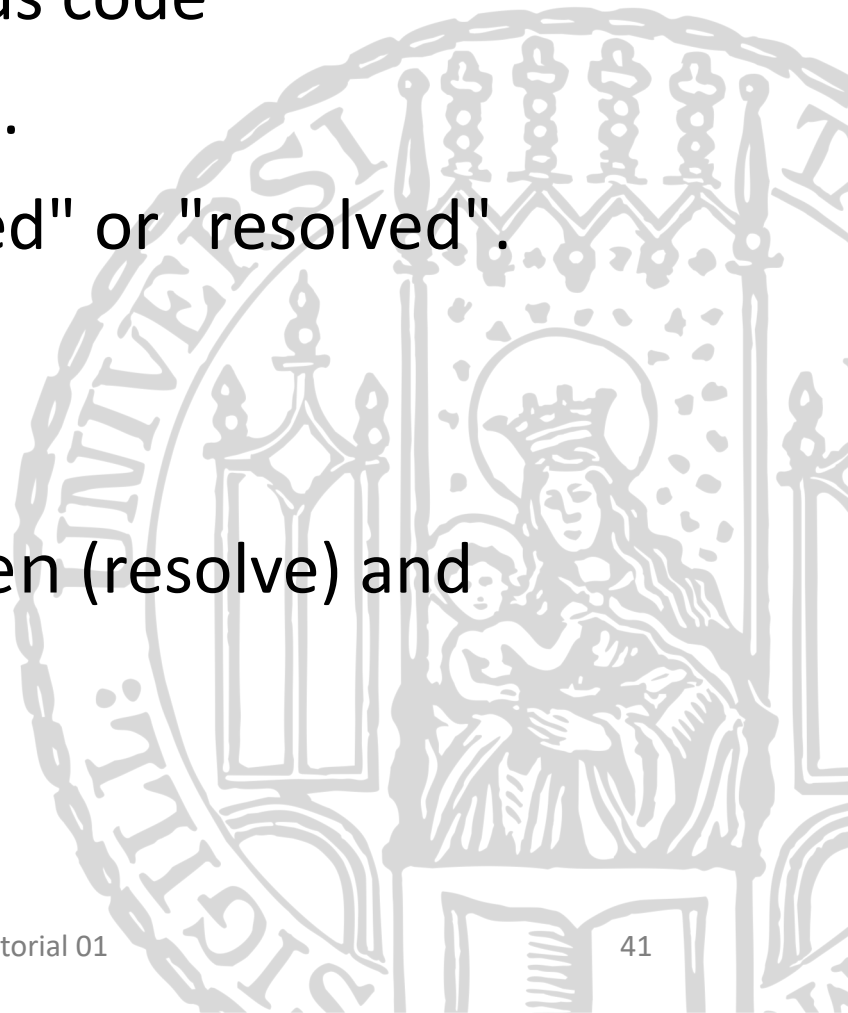
```
fs.readdir(source, function (err, files) {
    if (err) {
        console.log('Error finding files: ' + err)
    } else {
        files.forEach(function (filename, fileIndex) {
            console.log(filename)
            gm(source + filename).size(function (err, values) {
            if (err) {
                console.log('Error identifying file size: ' + err)
            } else {
                console.log(filename + ' : ' + values)
                aspect = (values.width / values.height)
                widths.forEach(function (width, widthIndex) {
                    height = Math.round(width / aspect)
                    console.log('resizing ' + filename + 'to ' + height + 'x' + height)
                        this.resize(width, height).write(dest + 'w' + width + filename, (err) => {
                            if (err)
                               console.log('Error writing file: ' + err)
                        })
                    }.bind(this))
                }
            })
        })
    }
})
```

http://callbackhell.com

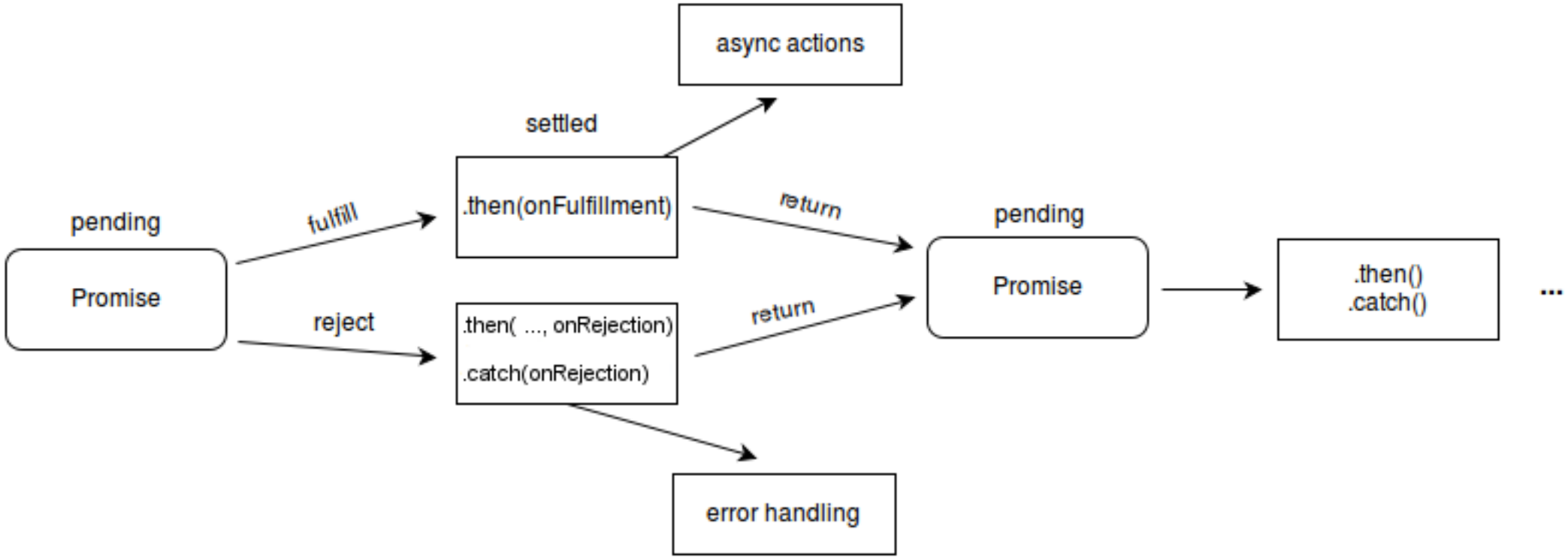
Promises

- Promises are a concise way to write asynchronous code
- Inside a promise, asynchronous code is executed.
- When the code completes the promise is "fulfilled" or "resolved".
- When the code fails the promise is "rejected".

- The result is used by callbacks attached via `.then` (resolve) and `.catch` (reject)
- Actions on promises can be chained



Promises



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

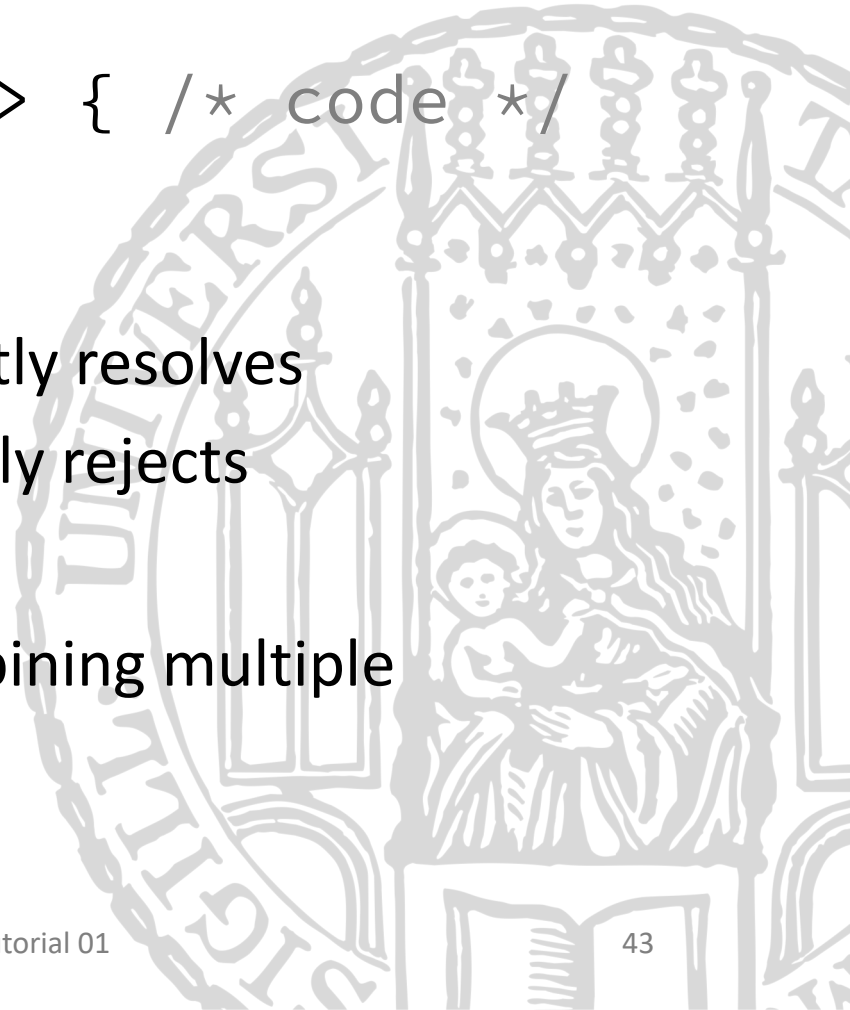
Promises

- Creating a promise:

```
new Promise((resolveCB, rejectCB) => { /* code */
});
```

- `Promise.resolve` creates a promise that instantly resolves
- `Promise.reject` creates a promise that instsantly rejects

- `Promise.all` / `Promise.race` allows combining multiple Promises.

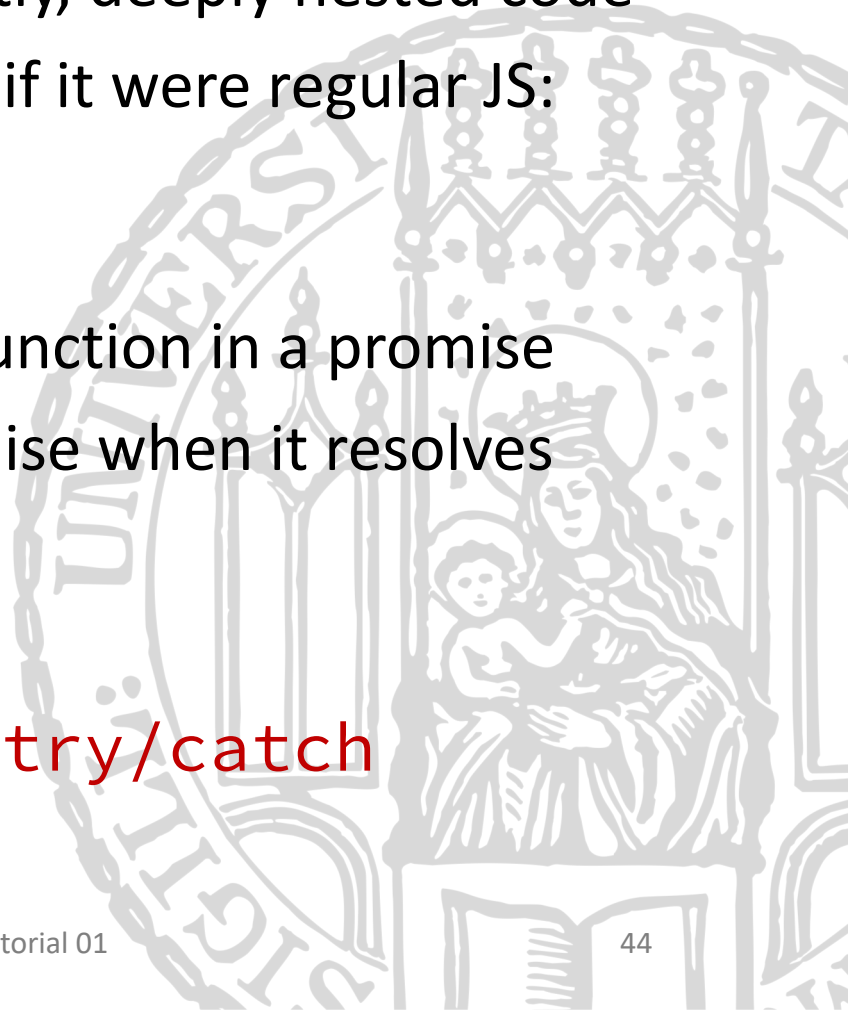


async & await

- Chaining multiple promises can lead to some unsightly, deeply nested code
- JS provides syntactic sugar to write Promise-code as if it were regular JS:
  `async & await`

- Making a function async "wraps" the result of that function in a promise
- Awaiting a promise "unwraps" the value in the promise when it resolves

- `await` can only be used in `async` functions
- Error handling in `async` functions can be done via `try/catch`



async & await

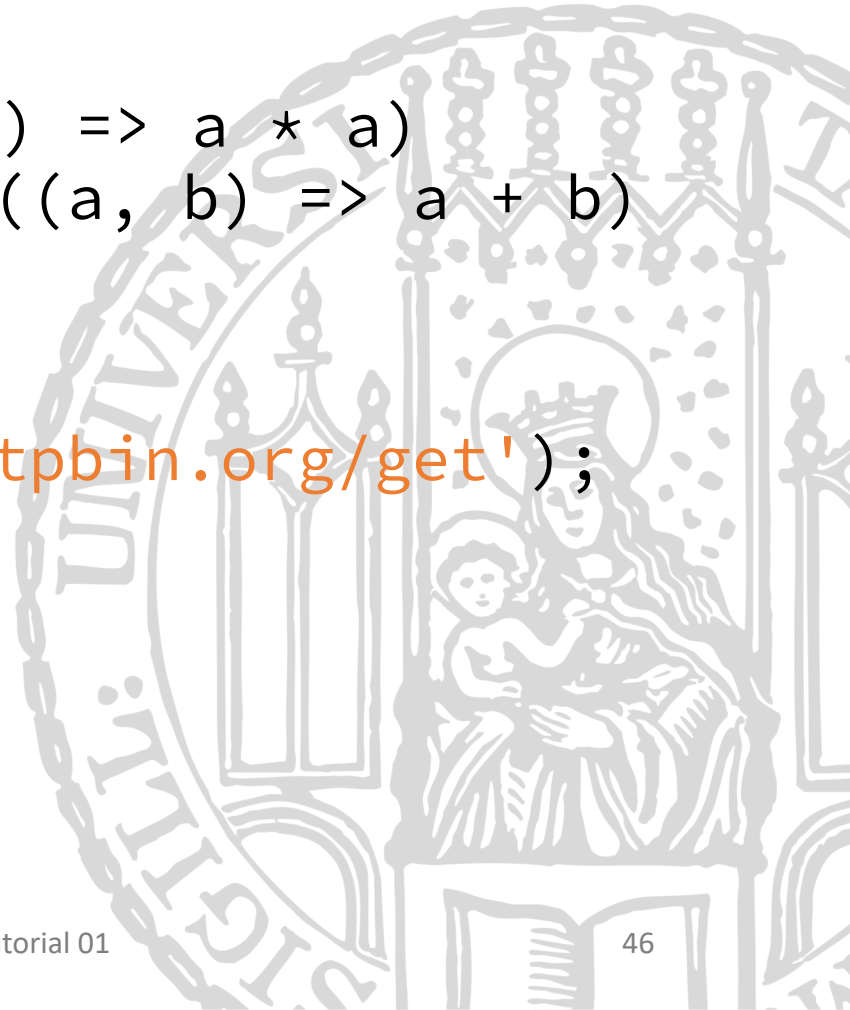
```
function () {
    return new Promise((resolve, reject) => {
        db.getByName(…)
            .then((id) => db.getById(id)
            .then((result) => resolve(result))
        )
    });
}

// The same function using async and await
async function () {
    const id = await db.getByName(…);
    const result = await db.getById(id);
    return result;
}
```

Roundup Quiz

- Describe the difference between `let` and `var`
- What is the result of `[ 1, 2, 3, 4 ].map((a) => a * a).reduce((a, b) => a + b)`
- Why will this fail:

```
function () {
  const value = await fetch('http://httpbin.org/get');
    .then((res) => res.json());
  return value;
}
```



Appendix

Object Destructuring

- Consider this less than ideal function:

```
function f (obj) {
  const foo = obj.foo;
  const bar = obj.bar;
  const baz = obj.baz;

  return 'Foo: ' + foo + ', Bar: ' + bar + ', Baz:' +
baz;
}
```

Object Destructuring

- We can improve that by object destructuring

```
function f (obj) {
  const { foo, bar, baz } = obj;

  return 'Foo: ' + foo + ', Bar: ' + bar + ', Baz:' + baz;
}
```

Object Destructuring

- Object destructuring can also happen in a functions parameters

```
function f ({ foo, bar, baz }) {
  return 'Foo: ' + foo + ', Bar: ' + bar + ',
Baz:' + baz;
}
```

Object Destructuring

- Object destructuring can also happen in a functions parameters

```
function f ({ foo, bar, baz }) {
  return 'Foo: ' + foo + ', Bar: ' + bar + ', Baz:' + baz;
}
```

Object destructuring can also be done with arrays:

```
const [ first, second ] = [ 1, 2 ];
```

even when the length does not fit

```
const [ first, second ] = [ 1, 2, 3, 4 ];
```

String Interpolation

- String interpolation allows writing concise string construction

```
function f ({ foo, bar, baz }) {
  return `Foo: ${foo}, ${bar}, ${baz}`;
}
```

Spread Operator

- Creating a new object from an old one:

```
const mmn = { name: 'MMN', lecturer: 'Prof. Hußmann' };

const mmn18 = { year: 2018 };
mmn18['name'] = mmn.name;
mmn18['lecturer'] = mmn.lecturer;
```

That is ugly. Modern JS can do better!

Spread Operator

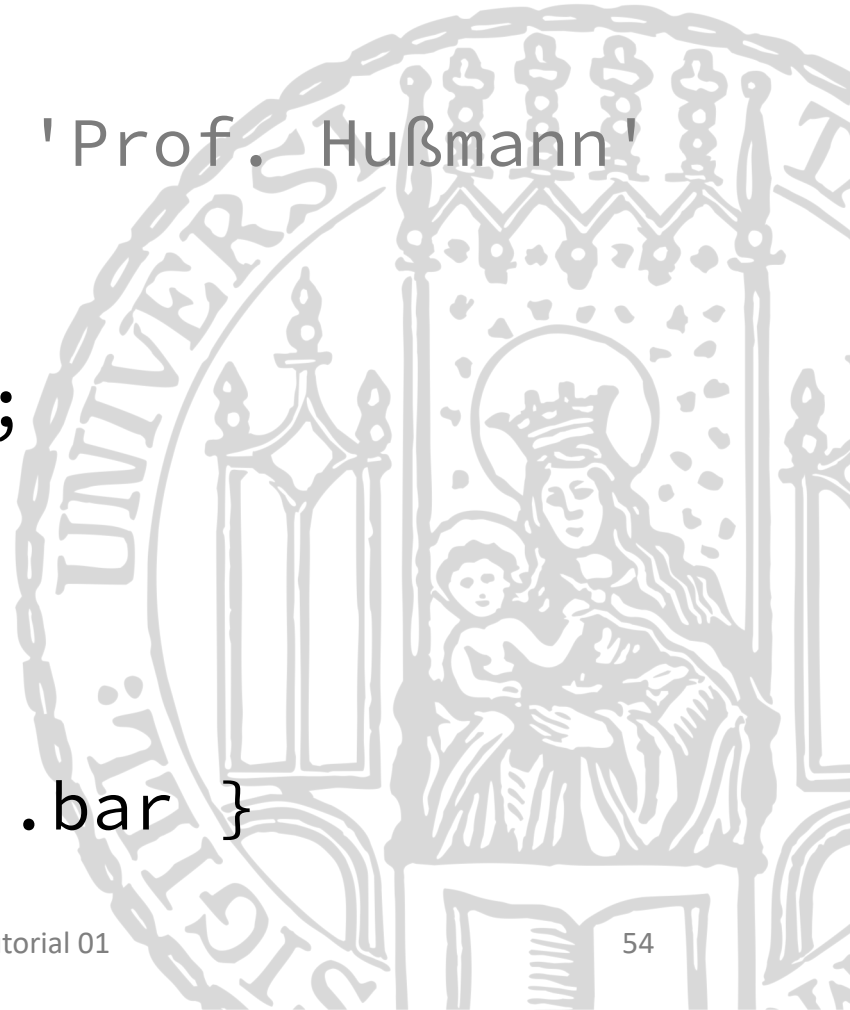
- Creating a new object from an old one:

```
const mmn = { name: 'MMN', lecturer: 'Prof. Hußmann'
};
```

```
const mmn18 = { ...mmn, year: 2018 };
```

That is it. Thanks to the Spread-Operator: `...`

You can also merge objects like this: `{ ...foo, ...bar }`

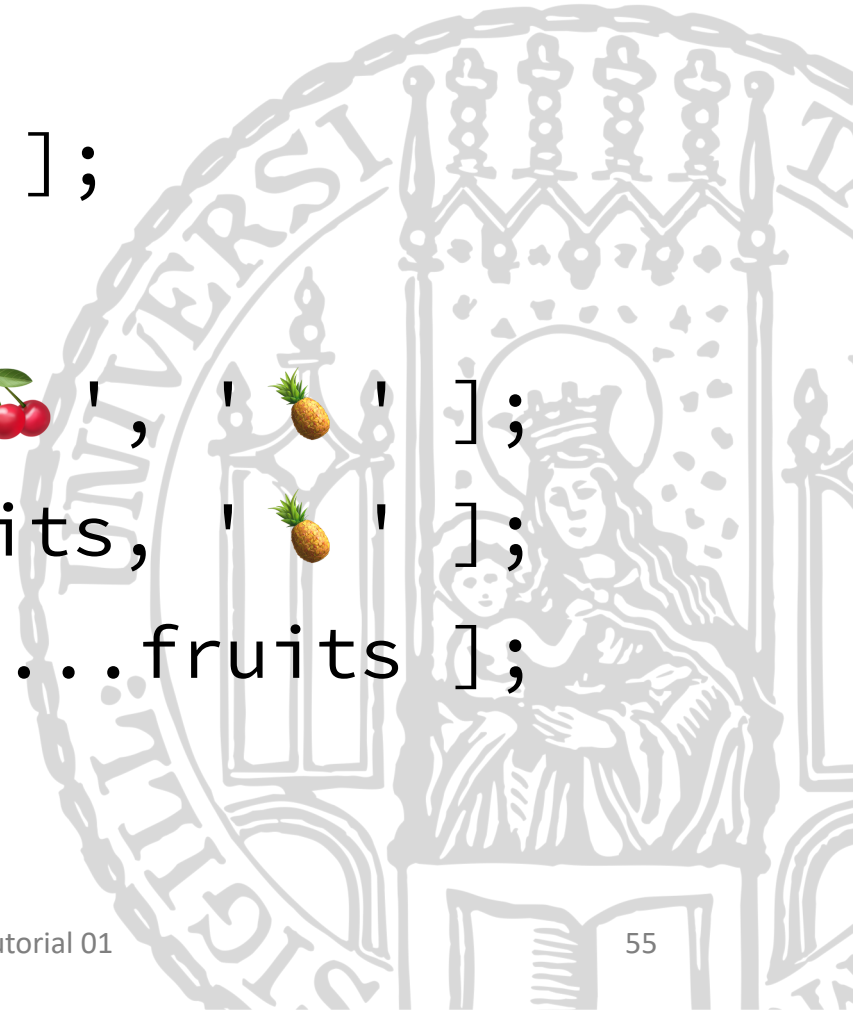


Spread Operator

- The spread operator also works for arrays:

```
const fruits = [ '🍏', '🍊', '🍇' ];

const moreFruits0 = [ ...fruits, '🍒', '🍍' ];
const moreFruits1 = [ '🍒', ...fruits, '🍍' ];
const moreFruits2 = [ '🍒', '🍍', ...fruits ];
```



Spread Operator

- With the spread operator, function can also have arbitary paramters:

```
function f (first, second, ...others) {
  return `${first}, ${second}, \
          and ${others.length} more.`;
}

f(1, 2, 3, 4, 5); // '1, 2, and 3 more.'
```

- All parameters after first and second are accessible as an array.

Advanced `console`

`console.info/.warn/.log/.error` are examples of functions with an arbitrary number of arguments.

While print debugging is not ideal, `console` can still be a good and convient tool for understanding ones code:

- `console.time/.timeEnd` allows timing code
- `console.trace` prints the stack trace for the currently running code
- `console.table` prints an array as table